K0: An investigation into using private tokens on smart contract platforms Version 0.1

Applied Blockchain

4 July 2019

Abstract

Smart contract blockchain platforms enable new kinds of decentralised applications used by mutually distrusting parties. However on most such platforms all transactional data is public, which renders them unusable for use cases requiring privacy. "Privacy coin" cryptocurrencies like Zerocash solve this problem for the money transfer use case by enabling private payments on a decentralised (special-purpose) blockchain. We describe an adaptation of this scheme for (general-purpose) smart contract blockchain platforms and propose a slight modification to the protocol to allow the use of private payments as triggers for actions in secondary smart contracts independent of the payment system itself.

1 Introduction

Blockchains provide a still relatively new paradigm of decentralised applications. Smart-contract platforms in particular allow the creation of a multitude of different types of applications between mutually distrusting parties, e.g. trading platforms or tracking systems for supply chains. However, the fact that in most implementations all the business transactions on a blockchain network are visible to all the members of the network has precluded many potential groups of users from adopting the technology, especially companies who wish to hide their commercial activities from their competitors. Research has shown that the public-key/address pseudonymity provided by most blockchain platforms does not provide much privacy at all because the transaction graph can be analysed. For the original blockchain use case of value transfer, recently this privacy problem has been attempted to be solved by so-called "privacy coins". It seems promising to try to bring privacy coin technology to the world of smart contract platforms and explore ways of augmenting smart contracts functionality with private payments. The goal is to enable arbitrary trustless *and* private blockchain applications to be built. This document describes the first iteration of one particular approach of doing that. We adapt the Zerocash scheme so that it can be used on a smart contract platform (with an Ethereum smart contract or a Hyperledger Fabric chaincode at its centre) and describe a way by which private payments in that scheme can be used to trigger actions in separate smart contracts. Please note that this is a rough high-level summary of an early-stage work-in-progress project. It should not be seen as a complete or even fully accurate representation of the codebase, nor as an academic paper in the field of cryptography. We tried to stay as close as possible to Zerocash and its implementation in Zcash, which has been in production for years and for which security proofs exist. However we do not make any claims about the security of the scheme described here at this point. Especially, it has not been security-audited.

2 Related work

This work is based on Zerocash [5] and its implementation in Zcash [9]. Miximus [4], ZETH [12] and Nightfall [10] are recently released schemes implementing Zerocash or parts of it on Ethereum as well. Zether [8] follows a different approach to providing confidential payments on smart contract platforms, as does AZTEC [13], which is specifically aimed at being usable and economic on the public Ethereum network. Cloak [1] is a protocol for private asset transfers and exchanges, while ZkVM [2] and ZEXE [7] are designs for more arbitrary private smart contracts.

3 High-level overview of Zerocash

The basis of our design is the decentralized anonymous payment (DAP) scheme described in the Zerocash paper [5] and its implementation in the first version ("Sprout") of Zcash [9]. We quickly summarise the scheme here and glance over some details for the sake of simplicity; for a more detailed description the reader is advised to consult the introductory section in [5]. The main state of the system is a list of note commitments (stored as a Merkle tree) and a list of "used serial numbers". The leaves of the Merkle tree represent notes or coins resembling UTXOs in Bitcoin, however while at least the values of the UTXOs are publicly visible in Bitcoin, here notes are binding and hiding commitments (ie. they are indistinguishable from random to observers unable to open the commitments) to the amount value of the note v, the owner's public key a_{pk} and a random value ρ (rho). Speaking abstractly, transactions take some of these notes as inputs and produce new output notes. The sum of the values of the inputs must equal the sum of the values of the outputs. Moreover, once a note is used as an input in a transaction, it is "spent" and cannot be used any more. To enable this in a private fashion, transactions do not reveal which input notes are spent. Transactions contain the "serial numbers" of the input notes (the serial numbers are the result of applying a pseudo-random function to ρ and the private key corresponding to the public key a_{pk}), the output note commitments, and a zero-knowledge proof. The proof attests that

- the creator of the transaction holds the private key corresponding to the input note public keys
- the owner knows the ρ value and the blinding factor r of each of the input notes
- the revealed serial numbers correspond to the input notes
- the input notes are included in the Merkle tree
- the sum of the input values is equal to the sum of the output values
- the output note commitments are well-formed, ie. each of them is a commitment to an amount, a newly created ρ and a public key

A transaction is valid if this proof can be verified and the serial numbers are not yet in the list of used serial numbers (which prevents double-spending). The result of a valid transaction is that the output note commitments are added to the note commitment Merkle tree and the serial numbers are added to the list of used serial numbers.

In order for the recipient of a payment to be able to spend the funds of an output note later on (by using it as an input in another transaction), they need to know (in addition to their private key) the amount v, the value ρ and the commitment blinding factor r. Therefore these are transmitted in encrypted form as part of the transaction.

Our contribution is an implementation of this scheme for smart contract platforms, which works with Ethereum and Hyperledger Fabric with small adjustments. Moreover, we add support for use cases where private payments on the system can be used as events to trigger actions in other smart contracts.

4 Description of our adaptation of Zerocash for smart contract platforms and our protocol modification

4.1 Components

We target smart contract platforms such as Ethereum [14] and Hyperledger Fabric [3]. In order to be able to discuss these more generally, we define two major abstract components:

- A *platform*, which can be seen as a "decentralised application server", but for the purposes of this description acts like a singleton entity. It holds the state (memory) for the scheme described here (in particular, the note commitment Merkle tree and the list of used serial numbers) and runs code ("smart contracts") triggered by function calls sent by the nodes. One important responsibility of the platform is to verify zero-knowledge proofs.
- *Nodes* which can send messages (function calls) to the platform and receive messages (events) from the platform. Typically every user of the system will have their own node. Nodes also need memory, to hold some state about the system and more importantly the user's secrets (private key and for each note ρ, v and r). In preparation to calling functions, nodes may generate zero-knowledge proofs.

A function call from a node triggers code on the platform to be executed, which may result in state changes and events broadcast to all the nodes. Every function call / state change / event emission cycle constitutes a *transaction*. Transactions are executed atomically and sequentially, usually the consensus mechanism of the blockchain will determine the order of transactions.

One important aspect of the smart contract platform is that it allows different types of "programs" (smart contracts) to be available in the system, and that smart contract code that gets executed during a transaction may itself send function call messages to other smart contracts, which will be processed in the same transaction.

The scheme discussed here assumes transactions can be sent anonymously and without cost. In private/consortium Ethereum setups this is usually not an issue, however in public Ethereum every transaction costs a certain amount of Ether, and this Ether has to be acquired from somewhere beforehand, which represents a challenge to anonymity. However there are ideas of overcoming these limitations, for example by the use of transaction relayers. In Hyperledger Fabric, relayers could be used as well, another option would be the use of the "Identity Mixer" technology.

4.2 Operations

The following describes the operations offered by the system. Most of the processes are almost identical to those of Zerocash and Zcash, therefore the reader is referred to [5] and [9] for a more detailed description.

4.2.1 Platform initialisation

The platform gets initialised with an empty note commitment Merkle tree and an empty list of used serial numbers.

4.2.2 Key generation

To participate in the system, a user needs to sample a random note private key (or, in Zcash parlance, spending key) a_{sk} from which the following keys are derived:

- $a_{pk} := H(a_{sk} \parallel 0^{256})$ (note public key or paying key)
- $sk_{enc} := SkEncGen(a_{sk})$ (encryption private key or receiving key)
- $pk_{enc} := PkEncGen(sk_{enc})$ (encryption public key or transmission key)

$SkEncGen$ is the private encryption key derivation function and $PkEncGen$ is the public encryption key derivation function (see section 4.4). a_{sk} and sk_{enc} must always be kept private (known only to the user). a_{pk} and pk_{enc} get concatenated to form a *K0 address*:

$$addr := a_{pk} \parallel pk_{enc}$$

4.2.3 Mint (Deposit)

As a store of value, the K0 system needs to provide a way to create or "mint" tokens. Viewed abstractly, the problem that needs to be solved is that a certain value needs to be added to a particular user's balance. How this is triggered differs in the implementations:

- In the Ethereum version the private token is "backed" by a regular ERC-20 token. A user is entitled to "mint" a certain amount of private coins by "locking up" an equivalent amount of the ERC-20 token in the smart contract (similarly she can later withdraw her ERC-20 token value again by "burning" an equivalent amount of private token value, see section 4.2.6). This mechanism essentially allows us to delegate the question of overall money supply and minting to the backing ERC-20 token.
- In the Fabric version (since there is no notion of smart contracts having native "accounts" which would allow a scheme of "locking up" value) one trusted "minter" organisation is entitled to mint all the value.

In both cases, the minting operation will increase the money supply by a publicly visible amount, and will ensure that this amount is subsequently spendable by a certain user whose identity is only known to the user herself and the "minter" (in the Ethereum version one would typically mint money for oneself). In particular, the minter samples a randomness ρ and creates a commitment k to ρ and the recipient's note public key a_{pk} using a hash function[1] and a blinding factor r[2]:

$$k := H(r \parallel [H(a_{pk} \parallel \rho)]_{128})$$

He then creates a commitment cm to k and the minted value v:

$$cm := H(k \parallel 0^{192} \parallel v)$$

The minter then sends v, k, cm and some encrypted $data$ to the smart contract in a call to the function $mint$:

$$mint(v, k, cm, data)$$

$data$ contains the values v[3], ρ and r encrypted with the recipient's encryption public key pk_{enc}[4]. The smart contract checks that cm equals $H(k \parallel v)$. If this check succeeds, it adds cm to a list of note commitments, stored as a Merkle tree, and emits a *Mint* event containing cm and $data$ as payload. Upon receiving this event, the recipient of the note will be able to decrypt $data$ using their secret key sk_{enc} and store the values v, ρ and r in a database that relates them to the note commitment cm and its position in the Merkle tree.

4.2.4 Transfer

A transfer operation "consumes" or "spends" two "input" notes and creates two new "output" notes[5]. The output notes will be added to the Merkle tree. It will not be revealed which of the available notes are used as the input notes, however for each of them their "serial number" will be revealed. The system keeps track of the serial numbers, which prevents double spending. Moreover the transfer operation will ensure that the user sending the transaction is actually entitled to spend the input notes, and that the sum of the values of the input notes matches the sum of the values

[1]Currently instantiated with the compression function of SHA256

[2]$[\cdot]_x$ denotes truncating a longer bit string to x bits.

[3]v is not strictly needed here because it is included in the public parameters to the function call, but is included for simplicity reasons as the same encryption/decryption code can be used for the transfer function (where v is hidden), see 4.2.4.

[4]There is no check in the system that ensures that data is actually the encryption of v, ρ and r. This means it is possible that some undecipherable data can be sent in this field. This will result in the recipient not being able to spend the note or even being notified of the payment. However when the intention is to "send someone money", there is no reason to do that. Should sender and recipient not want to trust the security of the encryption scheme or its implementation, they can choose to ignore this field and transmit the values through some other mechanism out-of-band.

[5]The restriction on two input notes and two output notes is due to the requirement of having a fixed circuit for the zero-knowledge proof.

of the output notes (ie. no money is created or lost in the process). In detail, a transfer operation is created in the following steps: A user (or rather, his wallet software) selects from his database two input notes for which he holds the private values v_1^{in}, ρ_1^{in}, r_1^{in} and v_2^{in}, ρ_2^{in}, r_2^{in}, respectively. He creates two output notes (typically one representing the intended payment to another user, and one "change" output for himself) by setting v_1^{out} and v_2^{out} and sampling ρ_1^{out}, ρ_2^{out}, r_1^{out} and r_2^{out}.

He creates a proof π that attests: Given Merkle tree root rt, values sn_1^{in}, sn_2^{in}, cm_1^{out}, cm_2^{out} and id_{callee}, I know input note info (a_{pk}^{in}, v_1^{in}, ρ_1^{in}, r_1^{in}, cm_1^{in}), (a_{pk}^{in}, v_2^{in}, ρ_2^{in}, r_2^{in}, cm_1^{in}), output note info ($a_{pk,1}^{out}$, v_1^{out}, ρ_1^{out}, r_1^{out}), ($a_{pk,2}^{out}$, v_1^{out}, ρ_1^{out}, r_1^{out}), Merkle tree paths $path_1$ and $path_2$, note private key a_{sk} and value $id_{callee}^{private}$ s.t.:

- The coins are well-formed:
  - $cm_1^{in} = H(H(r_1^{in} \parallel [H(a_{pk,1}^{in} \parallel \rho_1^{in})]_{128}) \parallel 0^{192} \parallel v_1^{in})$
  - $cm_2^{in} = H(H(r_2^{in} \parallel [H(a_{pk,2}^{in} \parallel \rho_2^{in})]_{128}) \parallel 0^{192} \parallel v_2^{in})$
  - $cm_1^{out} = H(H(r_1^{out} \parallel [H(a_{pk,1}^{out} \parallel \rho_1^{out})]_{128}) \parallel 0^{192} \parallel v_1^{out})$
  - $cm_2^{out} = H(H(r_2^{out} \parallel [H(a_{pk,2}^{out} \parallel \rho_2^{out})]_{128}) \parallel 0^{192} \parallel v_2^{out})$
- My private key corresponds to the public key of the input notes: $a_{pk}^{in} = H(a_{sk} \parallel 0^{256})$
- The serial numbers are computed correctly[6]:
  - $sn_1^{in} = H(a_{sk} \parallel 0 \parallel 1 \parallel [\rho_1^{in}]_{254})$
  - $sn_2^{in} = H(a_{sk} \parallel 0 \parallel 1 \parallel [\rho_2^{in}]_{254})$
- cm_1^{in} and cm_2^{in} are in the Merkle tree (ie. in a Merkle tree with root rt, $path_1$ is a path to leaf cm_1^{in} and $path_2$ is a path to leaf cm_2^{in})
- The value of id_{callee} is fixed for the proof: $id_{callee} = id_{callee}^{private}$ (these values will be needed for the "smart payment" feature described in section 4.2.5 and will be 0 for this use case)

The sender then calls the transfer smart contract function with the following call:

$$transfer(sn_1^{in}, sn_2^{in}, cm_1^{out}, cm_2^{out}, data_1^{out}, data_2^{out}, rt_{new}, id_{callee}, \pi).$$

$data_1^{out}$ and $data_2^{out}$ are the encrypted values v, ρ, r of the output values (see section 4.2.3). rt_{new} is the expected root of the Merkle tree after adding cm_1^{out} and cm_2^{out}. In this use case, id_{callee} will be set to 0 and therefore will be ignored by the smart contract. π is the proof described above. The smart contract will

- check that the serial numbers are not yet in the list of used serial numbers
- verify the proof with the current Merkle tree root rt and the sn values from the transaction as public inputs
- add the serial numbers to the list of used serial numbers
- add the note commitments to the Merkle tree
- emit a *Transfer* event with the payload $(sn_1^{in}, sn_2^{in}, cm_1^{out}, cm_2^{out}, data_1^{out}, data_2^{out})$.

Recipients will be able to decrypt and add to their database the v, ρ and r values encrypted in the respective *data* field of the *Transfer* event. The senders will be able to identify and mark the input notes as spent by the sn values.

[6] As pointed out in [12], the fact that only 254 bits of ρ are used here creates an attack vector which allows a sender to create multiple notes with the same serial number. We aim to address this issue in a future version of the protocol.

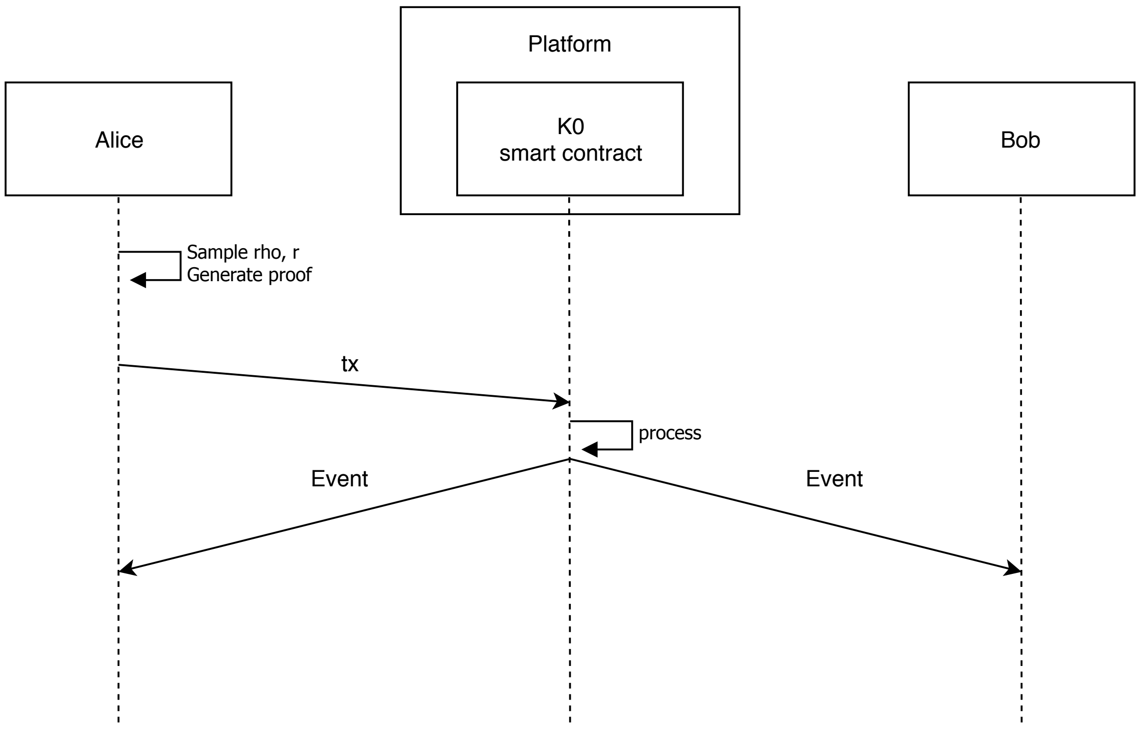

Figure 1: A payment flow.

4.2.5 "Smart payment"

Now let us assume a scenario where the recipient expects a certain payment (for example, a certain amount in exchange for some goods). In this case the party expecting the payment will typically communicate to the sender the value v and their public key a_{pk}. Recall that according to the transfer flow described in the previous section, the sender will then sample ρ and r, use these values to generate the payment and send them (and the amount v) to the recipient (in encrypted form as part of the transfer event). After this has happened, both sender and receiver have knowledge of v, ρ and r (but only the receiver will be able to spend the note using their private key). We can modify this process slightly by switching the responsibility of generating ρ and r from the sender to the recipient, ie. the party expecting the payment samples ρ and r, and sends v, ρ, r and a_{pk} to the sender. The sender will then use all these values in the payment. The result in terms of informational insight into the payment for the participating parties is the same, both know v, ρ and r, but only the recipient can spend the funds later using their private key. However this flow has the interesting property that the party expecting the payment can already precompute for themselves the cm value that will appear in the Merkle tree once the payment is added to the ledger. More importantly, in a smart contract system, they can register that particular cm in another smart contract. Later this smart contract can be convinced (for example through a proof asserting that the cm is included in the Merkle tree whose root rt the smart contract can query) that the payment has happened and, for instance, trigger some action for which the payment was a precondition. Lastly, if the action of informing a smart contract that a certain payment has happened and triggering it to act on it can be in included in the payment transaction itself, the payment and the action that is triggered by it can be modeled as an atomic operation. This is what the id_{callee} parameter in the transfer operation is for: If it is set, the system will call the function $handle(cm)$ on the smart contract specified by id_{callee} (in the Ethereum case, a contract address). The rest of the processing is similar to the normal payment case described in section 4.2.4. Note that the transaction is atomic, it will only have an effect if both the transfer and the triggered operation in the secondary contract succeed.

In the test and demo code for our current Ethereum implementation, this functionality is used to achieve a half-private atomic token transfer between the private payment token described here and

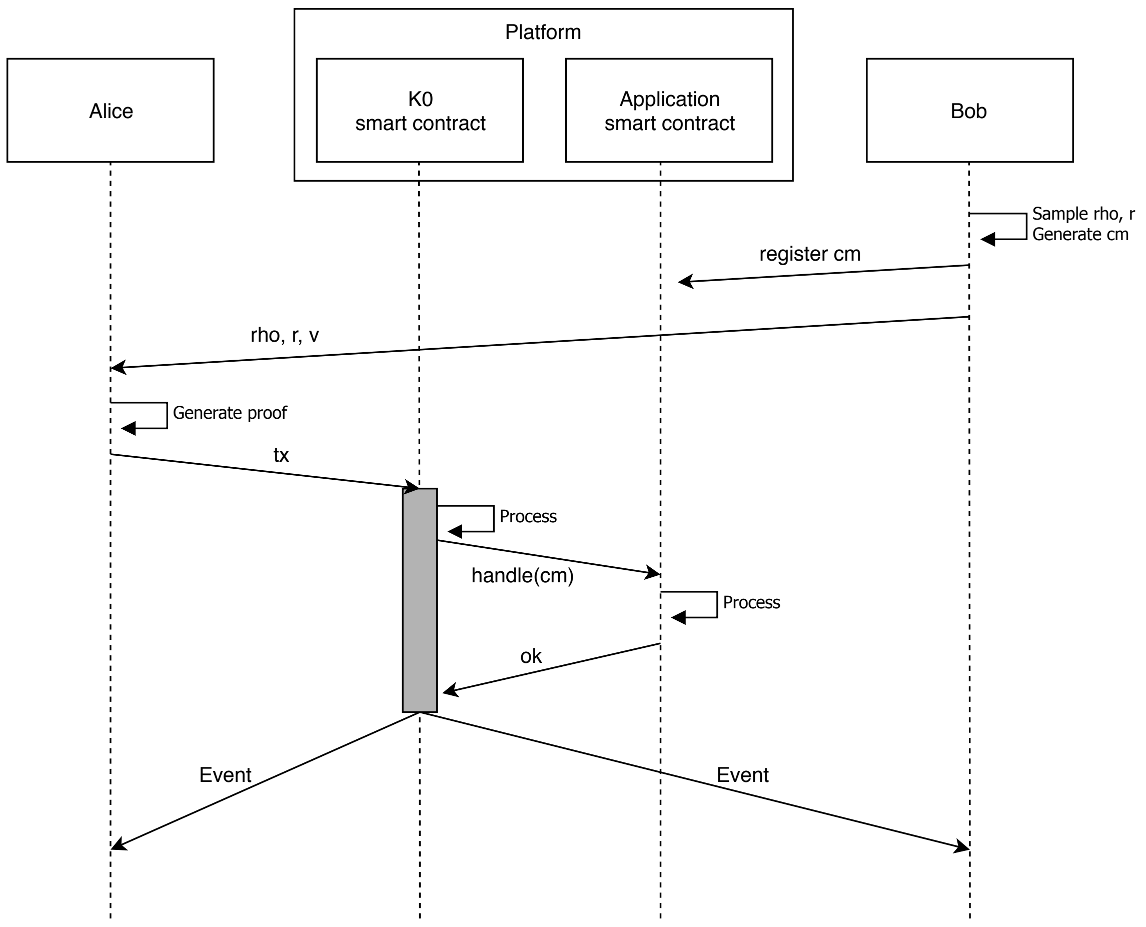

Figure 2: A "smart payment" flow.

a non-fungible ERC-721 token. The use case is that a holder of an instance of a certain ERC-721 token wants to offer that token instance (in our example, a "CarToken" representing a car, maybe in an on-chain game, or even a real-world "tokenised" car, for example in a trading application) for a certain amount of private token through a smart contract facilitating an atomic swap to a selected audience (without revealing anything about the monetary value to anybody else). First she samples ρ and r and generates the commitment cm using these values, the desired price v and her K0 address. She sets up a smart contract that will release the CarToken to the sender of a transaction resulting in cm and shares the values ρ, r and v with one or more potential buyers. She also needs to allow this smart contract to transfer the CarToken on her behalf (this functionality is part of the ERC-721 token standard). When the buyer sends the payment, he includes the car trading smart contract address in the id_{callee} field. While processing the transaction, the K0 token smart contract will call the function *handle* on the car trading contract, which will transfer the CarToken from the seller to the sender of the transaction, ie. the buyer. Note that this does not hide the identity of the traded CarToken. Also the Ethereum addresses of the old and new owner of the CarToken are public, however (assuming transactions can be sent anonymously, see section 4.1) the anonymity provided by the pseudonymity of the Ethereum addresses in this case is assumed to be much stronger than usual as there is no real "transaction graph" as the payments are hidden (especially if the adresses are *only* used to aqcuire, hold and dispose of the CarToken).

4.2.6 Burn (Withdraw)

The system allows users to reduce the overall money supply by "burning" notes, ie. spending a note (and revealing its serial number) without creating any new notes. In the Ethereum version backed by an ERC-20 token, this constitutes a withdrawal operation, ie. they will be able to redeem the equivalent amount of ERC-20 token from the pool of deposits. To burn a note, a user needs to call the smart contract function *burn*:

$$burn(v, sn, \pi)$$

v is the value of the note to be burned. sn is the serial number of the note to be burned. π is a proof attesting the following: Given values rt, v, sn and id_{sender} I know note info (a_{pk}, ρ, r, cm), private key a_{sk} and $id^{private}_{sender}$ s.t.

- $cm = H(H(r \,\|\, [H(a_pk \,\|\, \rho)]_{128}) \,\|\, 0^{192} \,\|\, v)$
- $a_{pk} = H(a_{sk} \,\|\, 0^{256})$
- $sn = H(a_sk \,\|\, 0 \,\|\, 1 \,\|\, [\rho]_{254})$
- cm appears in the Merkle tree with root rt
- $id_{sender} = id^{private}_{sender}$

id_{sender} is the sender of the transaction identified by the smart contract and will be added as a public input to the proof verification. It is included in the proof generation to prevent the usage of the proof in another transaction paying out to another user. If the proof verification succeeds, the smart contract adds sn to the list of serial numbers and decreases the overall money supply by v. In the Ethereum case it also transfers the ERC-20 value equivalent to v to the account of the sender of the transaction. A *Burn* event is emitted with sn as the payload. The sender's node software will be able to identify the input note by the sn and mark it as spent in the database.

4.3 Additional proofs

The current implementation deviates a bit from the scheme provided so far. This is due to decisions made to speed up the development process; it meant that we did not need to implement Merkle tree functionality on the smart contract platforms and were able to use the SHA256 compression function in our Merkle tree and the corresponding proofs. The modifications should not constitute a big qualitative difference and will probably be reverted in a later revision: The platform does not store the Merkle tree itself, but only its root rt. Nodes have to maintain a copy of the full Merkle tree and update it on each *Mint*/*Transfer*/*Burn* event emitted from the platform. During the mint operation, the transaction sender provides a proof π_1 that the addition of cm to the Merkle tree rt results in a new Merkle tree root rt'. Moreover, the outer commitment check ($cm = H(k \,\|\, 0^{192} \,\|\, v)$) is also provided as a proof, π_2. The full function call becomes $mint(v, k, cm, data, rt', \pi_1, \pi_2)$. If the proofs succeed, the smart contract updates the Merkle tree root stored in the contract: $rt := rt'$.

The transfer operation also contains a proof π_2 attesting that adding cm^{out}_1 and cm^{out}_2 to the the Merkle tree root results in a new Merkle tree root rt'. The full function call becomes

$$transfer(sn^{in}_1, sn^{in}_2, cm^{out}_1, cm^{out}_2, data^{out}_1, data^{out}_2, rt_{new}, id_{callee}, \pi_1, \pi_2).$$

where π_1 is the proof π described in section 4.2.4.

4.4 Encryption

The encryption scheme is copied and adapted from Zcash.

The function $Curve25519(n, q)$ denotes scalar multiplication of the Curve25519 point by the scalar n.

4.4.1 Private key derivation function SkEncGen

Analogous to section 4.2.1 in [9]:

- Start with private key sk_{enc} (see section 4.2.2).
- Apply a pseudo-random function: $sk_{enc}^{prfed} := H(a_sk \parallel 1 \parallel 0 \parallel 0^{254})$.
- Generate formatted private key $sk_{enc} := clamp_{Curve25519}(sk_{enc}^{prfed})$.
- Output sk_{enc}.

4.4.2 Public key derivation function PkEncGen

The public key derivation function $PkEncGen$ is $Curve25519$:

$$pk_{enc} := Curve25519(sk_{enc}, base_{Curve25519})$$

where $base_{Curve25519}$ is the Curve25519 base point.

4.4.3 Key agreement for symmetric encryption

The key agreement function is scalar multiplication of the public key by the private key (Diffie-Hellman): $secret_{DH} = Curve25519(sk_{enc}, pk_{enc})$.

4.4.4 Key derivation for symmetric encryption

The key derivation takes as input the Diffie-Hellman secret $secret_{DH}$, an ephemeral public key e_{pk}, and the recipient's public key pk_{enc}. The key K is the Blake2b-256 hash of the input $secret_{DH} \parallel e_{pk} \parallel pk_{enc}$ with the personalisation `"K0Cash"` $\parallel [0]^{80}$.

4.4.5 Symmetric encryption

The symmetric encryption scheme is authenticated encryption using `AEAD_CHACHA20_POLY1305` with empty "associated data", all-zero nonce $[0]^{96}$, and the 256-bit key K derived above.

4.5 Proving scheme

Like ZCash (Sprout), K0 uses the zero-knowledge proving scheme BCTV14a [6] implemented in the `libsnark` library [11]. Note that this scheme requires an initial "trusted setup" run by a trusted entity to generate the parameters for the proof generation and verification. It is of paramount importance that some data generated during the setup process ("toxic waste") is discarded or "forgotten", otherwise false proofs could be generated. Moreover, should the design of the "circuits" for which proofs are being created in the protocol change in the future, the setup needs to be repeated. There are other proving schemes available that mitigate some of these aspects which might be considered in a future version (see section 5). Another option would be to design the trusted setup as a multi-party computation (which ZCash did for both the initial Sprout circuit and the "Sapling" update).

5 Future work

This document describes the first iteration of our scheme. There are several measures in our roadmap to improve it, for example:

- Change of proving scheme to Groth16
- Change of hash function (Pedersen on babyJubJub)
- Keeping the full Merkle tree on-chain
- Addressing the "Faerie Gold attack" (section 8.4 in [9])

6 Conclusion

Standard smart contract platforms can run complex distributed applications but provide little privacy for the participants. The decentralised anonymous payment scheme introduced by Zerocash provides private value transfer. By implementing Zerocash for the use on smart contract platforms and making a small change to the protocol, we enable smart contracts to react to expected private payments. This is a first step in an exploration of ideas geared towards more privacy on smart contract platforms.

7 Acknowledgements

We thank Marc Beunardeau, Ariel Gabizon, Taylor Hornby, Harry Roberts, Antoine Rondelet and Kevaundray Wedderburn for their comments on drafts of this document and/or our approach in general. We thank the authors of Zerocash for the basis of this work and the Zcash team for their open-source code, which provided a very valuable reference for our implementation.

References

[1] Oleg Andreev. *Cloak*. URL: `https://github.com/stellar/slingshot/blob/main/spacesuit/spec.md` (visited on 2019-07-03) (cit. on p. 2).

[2] Oleg Andreev, Bob Glickstein, Vicki Niu, Tess Rinearson, Debnil Sur, and Cathie Yun. *ZkVM: Fast, Private, Flexible Blockchain Contracts*. May 2019. URL: `https://github.com/stellar/slingshot/files/3164245/zkvm-whitepaper-2019-05-09.pdf` (visited on 2019-07-03) (cit. on p. 2).

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains, v2*. April 17, 2018. arXiv: `1801.10228v2`. URL: `https://arxiv.org/abs/https://arxiv.org/abs/1801.10228v2` (cit. on p. 3).

[4] barryWhiteHat. *Miximus*. URL: `https://github.com/barryWhiteHat/miximus` (visited on 2019-07-03) (cit. on p. 2).

[5] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version)*. May 18, 2014. URL: `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf` (visited on 2019-06-26) (cit. on pp. 2, 3).

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture (updated version)*. February 5, 2019. URL: `https://eprint.iacr.org/2013/879` (cit. on p. 9).

[7] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. *ZEZE: Enabling Decentralized Private Computation.* 2018. URL: `https://eprint.iacr.org/2018/962` (cit. on p. 2).

[8] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. *Zether: Towards Privacy in a Smart Contract World.* URL: `https://crypto.stanford.edu/~buenz/papers/zether.pdf` (visited on 2019-07-03) (cit. on p. 2).

[9] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. *Zcash Protocol Specification, Version 2019.0.2.* June 18, 2019. URL: `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf` (visited on 2019-06-26) (cit. on pp. 2, 3, 9, 10).

[10] Chaitanya Konda, Michael Connor, Duncan Westland, Quentin Drouot, and Paul Brody. *Nightfall.* URL: `https://github.com/EYBlockchain/nightfall/blob/master/doc/whitepaper/nightfall-v1.pdf` (visited on 2019-07-03) (cit. on p. 2).

[11] SCIPR Lab. *libsnark: a C++ library for zkSNARK proofs.* URL: `https://github.com/scipr-lab/libsnark` (visited on 2019-07-03) (cit. on p. 9).

[12] Antoine Rondelet and Michal Zajac. *ZETH: On Integrating Zerocash on Ethereum.* April 4, 2019. arXiv: `1904.00905v2`. URL: `https://arxiv.org/abs/1904.00905v2` (cit. on pp. 2, 5).

[13] Zachary Williamson. *The AZTEC Protocol (version 1.0.1).* December 4, 2018. URL: `https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf` (visited on 2019-07-03) (cit. on p. 2).

[14] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger, Version d6ff64f.* June 13, 2019. URL: `https://ethereum.github.io/yellowpaper/paper.pdf` (visited on 2019-06-27) (cit. on p. 3).